THE APPLICATION OF MASSIVELY PARALLEL COMPUTATION TO INTEGRAL EQUATION MODELS OF ELECTROMAGNETIC SCATTERING

Tom Cwik, Robert van de Geijn[†] and Jean Patterson
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

[†]Department of Computer Sciences
University of Texas
Austin, TX 78660

Abstract

Integral equation methods are widely used in the analysis and design of electromagnetic systems. Traditionally, the limiting parts of the simulation have been the memory needed to store the dense matrix, and the computational time needed to solve the matrix equation. In this paper we report on the extension of integral equation solutions to new wavelength regimes, and completion of the solution in an amount of time that is practical for engineering applications. The numerical solution of the integral equation is computed on scalable, distributed memory parallel computers. Essential to the numerical solution was the development of a complex valued, highly optimized, dense matrix equation solution algorithm for scalable machines. A portion of the work outlined in this paper is the development of this production level library routine for the solution of linear equations on parallel computers, A convenient interface, useful for integral equation solutions among others, was specifically developed *in* this work. This algorithm has the conveniences offered by the sequential libraries, can be easily ported between parallel platforms, and has been placed in the public domain.

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology under a contract with the National Aeronautics and Space Administration.

1. Introduction

Integral equation methods are widely used in the analysis and design of electromagnetic systems. As witnessed by the wide range of problems solved over a period of nearly thirty years, these equations lend themselves to a convenient numerical solution [1]. Depending on the physical system considered, different integral equations can be written to model specific electromagnetic field or current components. For example, if the component being modeled consists of varying dielectric material such as the human body, a three-dimensional volume integral equation for the fields inside the body is written and solved. When the object is homogeneous or can be modeled as a perfect conductor, it is only necessary to write a three-dimensional surface integral equation for the fields, or equivalently induced currents, on the object's surface. The integral equation usually models the surface to better than 10 parts per electrical wavelength, and provides a stable solution for many types of components at many different electrical sizes. By using a moment method to solve the integral equation, a dense system of equations with corresponding matrix order proportional to the component's electrical size is computed, and its solution yields the induced current and secondary observational quantities. Traditionally, the limiting parts of the simulation have been the memory needed to store the dense matrix, and the computational time needed to solve the matrix equation. Because a large number of useful engineering components are many wavelengths in size, simulations using integral equation methods are fundamentally limited by the amount of available memory. In this paper we report on the extension of integral equation solutions to new wavelength regimes, and completion of the solution in an amount of time that is practical for engineering applications.

A numerical solution of the integral equation computed on scalable, distributed memory parallel computers will be outlined. Essential to the numerical solution was the development of a complex valued, highly optimized, dense matrix equation solution algorithm for scalable machines. This algorithm allows the efficient in-core solution of the linear system resulting from discretizing the integral equation. On conventional supercomputers, extensive libraries exist for solving the linear systems. Few, if any such libraries exist for parallel systems. Most routines that have been parallelized are research codes and have a cumbersome interface – if any. A portion of the work outlined in this paper is the development of a production level library routine for the solution of linear equations on parallel computers. A convenient interface, useful for integral equation solutions among others, was specifically developed in this work. This algorithm has the conveniences offered by the sequential libraries, can be easily ported between parallel platforms, and has been placed in the public domain.

2. The Electromagnetic Integral Equation

In this paper, the electric field integral equation (EFIE) for perfect conductors will be considered. This equation models a wide range of arbitrary objects including open and closed objects of arbitrary curvature and wires, and can be extended to handle surfaces with impedance coatings. The EFIE is derived directly from Maxwell's equations using the Green's function for an unbounded homogeneous space, and the boundary condition for the tangential electric field on a perfect electric conductor. The scatterer surface is denoted S and has an outward normal $\hat{n}$ (Figure 1). The total electric field E is broken into incident and scattered parts, where the incident field E^i is due to a time harmonic impressed source with time variation $\exp(j\omega t)$ radiating in the absence of the scatterer, and the scattered field E' is due to induced currents and charges on the scatterer radiating in free space. Derived using magnetic vector and scalar potentials, the scattered field is written as [2,3]

$$E'(r) = -jw\mu \mathbf{A}(\mathbf{r}) - \nabla\Phi(\mathbf{r}) \tag{1}$$

where the magnetic vector potential due to induced surface currents $\mathbf{J}(\mathbf{r}) = \hat{\mathbf{n}} \times \mathbf{H}(\mathbf{r})$ is

$$\mathbf{A}(\mathbf{r}) = \int_s \mathbf{J}(\mathbf{r}') \frac{e^{-jkR}}{4\pi R} dr' \tag{2}$$

and the scalar potential due to a surface charge $\sigma(\mathbf{r})$ is

$$\Phi(\mathbf{r}) = \frac{1}{\varepsilon} \int_s \sigma(\mathbf{r}') \frac{e^{-jkR}}{4\pi R} dr' . \tag{3}$$

Here, H is the total magnetic field, $R = |\mathbf{r} - \mathbf{r}'|$, and $\mathbf{r}$ and $\mathbf{r}'$ are arbitrary source and observation points. The surface charge is also related to the surface current through the relation

$$-jw\sigma = \nabla'_s \cdot \mathbf{J} \tag{4}$$

where ∇'_s represents the surface divergence operator in source coordinates. Enforcement of the perfect conducting boundary condition $\hat{\mathbf{n}} \times E = 0$ on the surface of the scatterer results in the EFIE

$$\hat{\mathbf{n}} \times (jw\mu\mathbf{A} + \nabla\Phi) = \hat{\mathbf{n}} \times \mathbf{E}^i \qquad r \in S \tag{5}$$

This is the equation that is discretized and solved using a method of moments solution. For the purposes of this paper, the associated magnetic field integral equation (MFIE) [2], combined field integral equation (CFIE) [4], or other related integral equations used for specific problems could be utilized in the following numerical solution.

3. Discretization of the Integral Equation

A solution for the unknown induced currents on the scatterer is found from a numerical solution of the EFIE (5). *PATCH,* a well developed code that solves (5) is used in this work [5, *6].* Initially, the scatterers surface is tessellated into triangular facets using standard geometry modelers and mesh generation packages. After replacing the surface charge in (3) by the surface divergence as outlined in (4), the induced currents on the surface facets are taken as the sole unknown in (5). A moment method is used to reduce the EFTE to a set of linear equations [7]. The order of the resulting matrix equation is the number of induced current functions.

In the ***PATCH*** code, the current existing on the scatterer is approximated by

$$\mathbf{J(r)} = \sum_{n=1}^{N} I_n \mathbf{f}_n (\mathrm{r}) \tag{6}$$

where f_n *are the* current expansion functions, I_n is the complex amplitude of the expansion function and N is the number of interior edges of the tessellated surface. A current expansion function consists of a pair of triangular facets attached at an edge [5]. This function has specific properties that make it useful for approximating an arbitrary current distribution on a general three-dimensional surface. It has the correct continuity of vector components across boundaries of mangle pairs; it properly models the surface charge in (3); and the use of triangular facets allows an arbitrary scatterer surface to be modeled.

Following the method of moments, a set of linear equations for the unknown amplitudes I_n is produced by testing (5) with an independent set of functions. The testing set is chosen to be identical to the current expansion functions, f_n *[5].* A symmetric inner product

$$\langle \mathbf{a}, \mathbf{b} \rangle = \int_{\partial v} \mathbf{a} \cdot \mathbf{b} ds \tag{7}$$

is defined, yielding the matrix equation

$$\langle j\omega\mu\hat{\mathbf{n}} \times \mathbf{A}, \mathbf{f}_m \rangle - \langle \hat{\mathbf{n}} \times \nabla\Phi, \mathbf{f}_m \rangle = \langle \hat{\mathbf{n}} \times \mathbf{E}^i, \mathbf{f}_m \rangle \quad r \in S \tag{8}$$

By testing with the same N functions used in (6), a NxN set of equations results

$$\mathbf{ZI} = \mathbf{V} \tag{9}$$

where Z is the impedance matrix, I is the vector of unknown current amplitudes, and V is the excitation vector.

The numerical solution of (9) generally proceeds by computing the impedance matrix elements, factoring it into a LU decomposition, and solving for the induced current given an excitation. Observable quantities such as the radar cross section, are then computed from the known current amplitudes. When using a shared memory machine, the impedance matrix is stored in memory and operated upon. When a distributed memory machine is used, the matrix must be broken into pieces and distributed across the memory of all processors in use. The manner in which the matrix is decomposed is dependent upon the matrix solution algorithm. General information on the description and application of distributed memory, parallel computer architecture for electromagnetic computation can be found in [8, 9].

4. The Parallel Dense Matrix Equation Solver

In this section, we describe a state-of-the-art parallel dense matrix solver routine developed during the course of this work. This routine is very similar to those included in a prototype ScaLAPACK library [10]. The main advantages of this routine area friendly user interface and an efficient and highly accurate condition number estimator.

In the design of this parallel dense matrix equation solver, the following issues were taken into account:

- Hierarchical memory on each node; the amount of memory and its organization on a processor must be take into account to achieve high performance [11,12].

- Data decomposition resulting from the integral equation formulation; the algorithm which calculates the matrix entries may lead to a natural decomposition of the matrix among the processors.

- Data decomposition that leads to an efficient matrix factorization; the factorization algorithm may require a specific decomposition of the matrix among the processors for high performance. This decomposition will be based on communication and scalability issues. In general this decomposition and the decomposition due to the integral equation will not coincide. Indeed, a usable and efficient interface between the parallel matrix equation solver, and the matrix computation segments of the code are essential.

- Robust solution of the matrix equation; the solver must produce stable results and yield an estimate of the stability of the system.

The first issue requires us to start with an approach that has merit on a conventional supercomputer. The second and third indicates the possible conflict between the data decomposition required by the application versus the data decomposition that leads to optimal performance of the dense matrix equation solver. The robustness of the solver indicates that the problem may yield a matrix that requires pivoting to produce an accurate result. Moreover, the application may need to know information about the conditioning of the linear system. We will address each of these issues in subsequent sections.

4.1 Optimal use of Caches

Considered solving the dense linear system (9), written as

$$Ax=b \tag{1o}$$

where $A \in C^{n \times n}$. Ignoring for the moment the use of pivoting for stability, it is customary to perform this operation by first computing its LU factorization

$$A = LU \tag{11}$$

where L *is* lower triangular with unit diagonal, and U is upper triangular. The system can then be solved by computing

$$Ly=b \tag{12}$$

followed by

$$Ux=y \tag{13}$$

In the past, this could be easily accomplished by calling the appropriate LINPACK [13] library routines.

On a typical processor with hierarchical memory, the LINPACK routine will not perform anywhere near the processor's peak rate. One reason is that the implementation is rich in vector-vector operations, which makes access to main memory a bottleneck. In essence, O(n) operations are performed on $O(n)$ data, and the processor cache, which allows high data access rates, cannot be utilized optimally [12].

LAPACK [14] corrects this by reformulating the factorization in terms of matrix-matrix operations. Partition A, L and U as follows

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} \tag{14}$$

#

where $A_{11}, L_{11}, U_{11} \in C^{k \times k}$. This leads to the equations

$$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} U_{11} \tag{15}$$

$$A_{12} = L_{11}U_{12} \tag{16}$$

$$A_{22} - L_{21}U_{12} \cdot L_{22}U_{22} \tag{17}$$

We see that the LU factorization can be formulated by overwriting a panel of width k with its LU factorization (15), followed by solving the triangular system with multiple right-hand-sides (16). Finally, the bulk of computation is in updating A_{22}, using a matrix-matrix multiplication, followed by a recursive factorization of the result (17). During the update of A_{22}, $2k(n-k)^2$ operations are performed on $(n+k)(n-k)$ data, allowing data to be brought into cache, after which a large amount of computation occurs before it is returned to memory. This overcomes the memory access bottleneck.

4.2 Parallel Implementation

Now that we have identified an appropriate approach for a sequential algorithm, it must be parallelized. However, we must keep in mind that the resulting algorithm will be called after the matrix has been calculated. Hence, we will first discuss how a user may wish to view the assignment of portions of the matrix to processors.

Assume that the physical processors, $\mathbf{P}_{ij}$ form a logical two dimensional $p_r \times p_c$ array, where i and j refer to the row and column index of the processor. Then the simplest assignment of matrix elements to processors is to partition the matrix

$$A = \left(\begin{array}{c|c|c} B_{00} & \cdots & B_{0(p_c-1)} \\ \hline \vdots & & \vdots \\ \hline B_{(p_r-1)0} & \cdots & B_{(p_r-1)(p_c-1)} \end{array} \right) \qquad (18)$$

where $B_{ij} \in C^{m_i \times n_j}$, $m_i \approx n/p_r$, and $n_j \approx n/p_c$. Submatrix B_{ij} *is* then assigned to processor $\mathbf{P}_{ij}$.

A straight forward parallel LU factorization, again ignoring pivoting, can now proceed as follows: Assume in (14) the column panel consisting of A_{11} and A_{21} resides within a single column of processors, while the row panel consisting of A_{11} and A_{12} resides within a single row of processors. The process starts by having processors that hold portions of the column panel collaborate to factor that panel. Next, the result can be broadcast within rows of processors to the other columns. The row of processors that holds A_{21} can then in parallel update this panel. Finally, A_{22} must be overwritten by $A_{22} - L_{21}U_{12}$. Concentrating on an arbitrary processor $\mathbf{P}_{ij}$, observe that only portions of L_{21} that reside in row i must be present on this processor, as well as portions of U_{12} that reside in column j. The former submatrix is already on the processor due to the broadcast of the factored panel within rows. The latter can be brought to the appropriate processors by broadcasting portions of U_{12} within columns of processors.

While the above scheme is straight forward, it can be easily seen that it does not yield a well distributed workload. Indeed, as the algorithm recursively proceeds with updated submatrix A_{22}, eventually processors become idle, until eventually only one processor is busy. This can be overcome by blocking the matrix into much smaller k x k submatrices, and *wrapping these* in two dimensions onto the logical processor array. In other words, partition

$$A = \left(\begin{array}{c|c|c} C_{00} & \cdots & C_{0(N-1)} \\ \hline \vdots & & \vdots \\ \hline C_{(M-1)0} & \cdots & C_{(M-1)(N-1)} \end{array} \right) \tag{19}$$

where $M \approx m / k$, and $N \approx n / k$ and all blocks are of size k x k. Then block C_{ij} is assigned to processor $\mathbf{P}_{(i \bmod p_r)(j \bmod p_c)}$. The algorithm proceeds as before, but as the problem deflates, processors continue to participate. Adding pivoting to this approach is relatively straight-forward [15,10,16].

4.3 Interface

Clearly, it is not reasonable to require the matrix fill portion of the application to generate the matrix in wrapped form. We now show that the matrix can be generated in the more convenient form given in (18), while the computation can assume it is wrapped as in (19), *without any need to perform communication.*

Consider what would happen if the matrix is generated as in (18) by the application, after which the library routine is called that assumes the matrix was wrapped as in (19). It is not hard to see that in effect, the LU factorization computed as a result is that of a permuted matrix

$$\tilde{L}\tilde{U} = \tilde{A} = P_{left} A P_{right} \tag{20}$$

However, we are not interested in the factorization. We are interested in solving the system Ax= b. It is natural to assume that vector b is distributed like a column of A, while the solution vector is distributed like a row of A. After all, elements b correspond to rows of A, while elements of x correspond to columns of A. *If* these vectors are distributed con formal to the original matrix A, *the* solver will view the right hand side as a permuted vector $\tilde{b} = P_{left} b$. It will proceed to solve the system

$$\tilde{A}\tilde{x} = \tilde{b} \tag{21}$$

leaving the result vector $\tilde{x}$ wrapped onto a row of processors, passing this solution back to the application.

Finally, the application views the resulting vector as the actual solution x, distributed among a row, but not wrapped. This is equivalent to viewing $P_{right}^{-1}\tilde{x}$ as the solution vector x. However,

$$\tilde{A}\tilde{x} = \tilde{b} \tag{22}$$

is equivalent to

$$\left(P_{left}AP_{right}\right)\tilde{x} = P_{left}b \tag{23}$$

and

$$A\left(P_{right}\tilde{x}\right) = b \tag{24}$$

Notice that while the solver views both matrix and vectors as being wrapped and the matrix fill algorithm views them as blocked, the resulting interpretation of the answer yields a correct solution vector.

4.4 Condition Number Estimation

The condition number estimator provided by LINPACK is generally regarded as a good method for obtaining a useful lower bound for estimating the one-norm of a given matrix. Once the matrix is factored, the cost of this estimate is the computation of a few triangular solves, and therefore of little consequence for large matrices. More recently, a better estimator was proposed by Nick Higham [17]. Generally, the estimate returned is accurate practically to machine precision. While more expensive than the LINPACK estimator, the cost remains on the order of that of a number of triangular solves. It is this estimator that has been incorporated into LAPACK. Conveniently, it is the LAPACK-style condition number estimator that parallelizes better. Ignoring the possibility of overflow for very ill-conditioned matrices which a full implementation of the LAPACK estimator would guard against, this estimator becomes a sequence of parallel triangular solves. It is this algorithm that has been incorporated into our solver. Details go beyond the scope of this paper.

4.S Scalability and Performance

The data decomposition used by the solver is specifically designed to yield high performance. Moreover, it is designed so that this performance scales well with the number of processors of the parallel computer. While details of the analysis of the implementation is clearly beyond the scope of this paper, we will sketch the implications of the design issues below. The interested reader can find details in [10].

As noted previously, the scattering problem we are solving is of sufficient complexity that the problem size is limited by the capabilities of the computer being used, at least currently. In our case, the limitation is the size of the aggregate memory of the parallel computer. As a result, we would hope that a large parallel system being used for the largest problem that fits in its memory, would be as efficient as a similar, but smaller parallel system when solving the largest problem it is capable of handling. The analysis in [10] shows that for the approach taken here, efficiency can be maintained as long as the matrix dimension increases with the square-root of the number of processing nodes. Since storage requirements grow as the matrix dimension squared and as the square-root of the number of processors, these requirements are met. This property is illustrated by the experimental data obtained on the Intel Touchstone Delta. In Figure 2, we see that as the number of nodes is increased, the problem size increases, but the efficiency, or performance per processor, remains constant.

Figure 3 shows the overall performance attained by our linear equation solver as a function of problem size for different numbers of processors. Figure 4 shows the similar result for CPU time to solve the linear system. The largest matrix system (n= 16,200) took less than 19 minutes to solve on 512 processors of the Touchstone Delta. It should be noted that the peak performance of each individual processor is in the 35-40 MFLOPS range for this kind of problem, indicating we are achieving better than 50% of peak for this part of the computation. Moreover, all performance graphs include the condition number estimation, and forward and backward substitution phases of the linear solve, which do not generally perform well on parallel computers.

5. Parallel Computation of Matrix Elements

As outlined in Section 4.3, the matrix equation solution algorithm operates on matrix data that has been mapped to the processors memory in a specific manner. This algorithm requires that contiguous blocks of the matrix be stored in each processors memory. When viewed as a library routine, it is of no consequence that the algorithm operates on a permuted version of the impedance matrix, since the resultant current vector is returned correctly ordered. It *is* essential though, that the algorithm operates on the wrapped version of the matrix when performing the matrix factorization and solution, since without this wrapping the solvers performance would greatly suffer.

Each processor calculates its piece of the impedance matrix as shown in (18). The parallel machine is viewed as a two-dimensional grid of processors – typically, a ratio of 1 row to 8 columns is found to give optimum performance. On a 512 processor machine, the grid is therefore 8 by 64 resulting in blocks of the matrix being approximately *n/8* rows by *n/64* columns in size. Since the matrix order will not in general be a multiple of the number of processor rows and columns, blocks in the bottom row and right-most column will be slightly smaller in size. When computing matrix elements, each processor only needs to calculate those rows and columns corresponding to its block, skipping the other matrix element calculations. In the *PATCH* code, the current basis functions associated with a triangular patch correspond to the columns of the impedance matrix and those of the testing patches correspond to the rows.

Ideally, parallel computation of the matrix elements would proceed by looping over row and column indices, performing the integrations necessary to compute matrix elements in the block of elements that reside in each processor. In the original *PATCH code,* to speed the sequential execution of the fill algorithm, it was initially recognized that calculations performed at the center of a patch are common to its three edges. It was efficient to compute the integrals associated with a triangular patch and distribute the results *over* elements of the matrix associated with the three *edges* of the patch. The matrix fill algorithm of *PATCH* therefore loops over pairs of current and match triangular patches. In the parallel computation of the matrix elements, an amount of redundancy develops at this stage since the three edges of both the source and testing patch - corresponding to row and column indices of a matrix entry - will not generally reside in one processor. The calculations not contributing to the edges associated with the processor performing calculations are therefore thrown out since they are being duplicated in the processors where they are used. This redundancy can be removed by performing the calculations once, communicating the matrix entry to its correct processor and storing it in its proper location within the matrix block. By performing this communication, no calculations would be repeated, though extra time would be needed for communicating the data. Because of this unknown overhead, and because of the added complexity in coding, this step was not taken. It is noted that if a different integral equation formulation results in matrix elements that have no calculations in common, this redundancy is naturally removed.

6. Parallel Computation of Observable

With the major part of computations and parallelization completed, all that remains is to calculate observational quantities associated with the scatterer. In PATCH, these are mainly the near and far-fields, and associated radar cross section. Field calculations are completed by performing the forward integration of the now-known currents, and Green's function evaluated at given near or far-field observation points. Because of the discretization of the current in (6), this integration results in a sum of the individual field components due to each current basis function. The parallelization of this integration is completed by breaking the sum into parts equally distributed over all processors. Since the solution vector has been distributed to all processors, the decomposition is rudimentary – each processor calculates field components due to current basis functions associated with a set of triangular patches. The partial sums are then added together to produce the field or cross section at a specific observation point.

7. Performance and Results

With the matrix equation solver fully integrated into the *PATCH code,* results can be obtained for various size problems run on different machine sizes. The following results were made on the Intel Touchstone Delta. A perfectly conducting cube scatterer is modeled with varying mesh densities. Two types of results were tabulated. First, results for scaled problem sizes are shown in Figure 5. In this case, the size of problem modeled - as represented by the matrix order is increased proportionally with the number of processors used. Components of the code are broken out of the total time and displayed in Figure 5. It is seen that the computation time is dominated by the matrix fill and factor portions of the computation. The solution was found for a single excitation, and the radar cross section was calculated in three planes in one degree increments. For the largest size problem that fits in 512 processors, total execution time required was 39.4 minutes.

The second set of results are for fixed size problems. Three different problem sizes are considered -3042, 8712, and 12,168 unknowns. For each size, the problem is run on the smallest sized machine it fits on, as well as larger machines. As seen from Figure 6, total execution time decreases as the fixed size problem is run on larger machines. A point of diminishing returns is reached where the use of a larger machine will not significantly decrease solution time. This point has been reached for the case of 3042 unknowns, but not yet for the larger sizes.

In Figure 7 the computed radar cross section is compared against measurements for a cube of side 5 wavelengths [18]. The total surface area modeled is 150 square wavelengths since no symmetry was utilized; 16,200 unknowns were used to model the induced current. Results in three planes are shown for broadside incidence.

8. Summary

This paper has outlined a numerical solution to integral equations completed on massively parallel computers. Essential to this numerical solution was the development of a parallel, high performance, dense matrix equation solution algorithm. Key to this development was an efficient and simple interface to the matrix fill portion of the integral equation code. This parallel routine can be obtained via anonymous ftp at *cs.utexas.edu*, in directory pub/rvdg/complex. solver, as well as from the electromagnetic software library EMLIB at *microwave.jpl.nasa. gov*, *in* directory pub/parallel/complex. solver.

Because of the large disk storage available on parallel computer systems, a natural extension of the above work is an out-of-core algorithm for use with integral equation codes. Initial work in this area was described in [19] where a 48,672 order matrix equation resulting from the EFIE was solved. Existing systems will allow for the storage of matrices of nearly twice this dimension. Current effort is underway to produce efficient and highly performing out-of-core algorithms similar to those reported in this paper.

Acknowledgments

This project was supported in part by Intel SSD, in part by DARPA under contract number DAAL03-91-C-0047 administered by ARO, and in part by DOE under contract number DE-AC05-840 R21400. This research was performed in part using the Intel Touchstone Delta System

operated by Caltech on behalf of the Concurrent S upercomputing Consortium. Access to this facility was provided by NASA.

References

[1] E. K. Miller, A selective survey of computational electromagnetic, *IEEE Trans. APS,* vol. *M-36,* pp. 1281-1305, Sept. 1988.

[2] A. J. Poggio and E. K. Miller, Integral equation solutions of three-dimensional scattering problems, in Computer *Techniques for Electromagnetic,* R. Mittra Editor, Hemisphere Publishing Corp., New York, 1987.

[3] R. F. Barrington, *Time Harmonic Electromagnetic Fields,* McGraw-Hill, New York, 1961.

[4] J. R. Mautz and R. F. Barrington, A combined-source solution for radiation and scattering from a perfectly conducting body, *IEEE Trans. APS,* vol. AP-27, pp. 445-454, July, 1979.

[5] S. M. Rae, D. R. Wilton, and A. W. Glisson, Electromagnetic scattering by surfaces of arbitrary shape, *IEEE Trans, AP-30,* pp. 409-418, May, 1982.

[6] W. Johnson, D. R. Wilton, and R. M. Sharpe, Modeling scattering from and radiation by arbitrary shaped objects with the electric field integral equation triangular surface patch code, *Electromagnetic,* vol. 10, pp. 41-64, Oct., 1990.

[7] R. F. Barrington, *Field Computation by Moment Methods,* IEEE Press, New York, 1993.

[8] T. Cwik and J. Patterson, Editors, Computational electromagnetic and supercomputer architecture, *Progress in Eelctromagnetics Research, vol. 7,* EMW Press, Boston, In Press.

[9] T. Cwik, Parallel decomposition methods for the solution of electromagnetic scattering problems, *Electromagnetic, vol. 12,* pp. 343-357, Dec. 1992.

[10] J. J. Dongarra, R.. van de Geijn, and David Walker, A look at scalable dense linear algebra libraries, *Proceedings of Scalable High Performance Concurrent Computing '92,* April 27-29, 1992.

[11] J. J. Dongarra, J. Du Croz, S. Hammarling and I. Duff, A Set of Level 3 Basic Linear Algebra Subprograms, *Trans. Math. Soft.,* vol. 16, No. 1, pp. 1-17, 1990.

[12] J. J. Dongarra, I. S. Duff, D. C. Sorenson, and H. A. van der Vorst, *Solving linear systems on vector and shared memory computers,* SIAM, Philadelphia, 1991.

[13] J. J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart, LINPACK User's Guide, SIAM, Philadelphia, 1979.

[14] E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen, *LAPACK Users' Guide,* SIAM, Philadelphia, 1992.

[15] R. van de Geijn, Massively parallel LINPACK benchmark on the Intel Touchstone DELTA and iPSC/860 systems, *Dept. Comp. Sci., U of Texas, TR-91-28:, 1991.*

[16] R. van de Geijn, Dense Linear Solve on the Intel Touchstone Delta System, *CompCon92, 37th IEEE Computer Society International Conference, Feb. 24--28,* 1992.

[17] N.J. Higham, FORTRAN Codes for Estimating the One-Norm of a Real or Complex Marnx, with Applications to Condition Estimation, *ACM Trans. on Math. Software,* Vol. 14, No, 4, pp. 381-396, 1988.

[18] M. G. Cote, M. B. Woodworth, and A. D. Yaghjian, Scattering from the Perfectly Conducting Cube, *IEEE Trans AP-36*, pp. 1321-1329, Sept. 1988.

[19] T. Cwik, J. Patterson, and D. Scott, "Electromagnetic Scattering Calculations on the Intel Touchstone Delta," *Proc. IEEE Supercomputing 92,* Minneapolis MN, pp 538-542, Nov 16-20 1993. (Gordon Bell Prize Finalist paper also summarized in *IEEE Computer,* Jan 1993.)

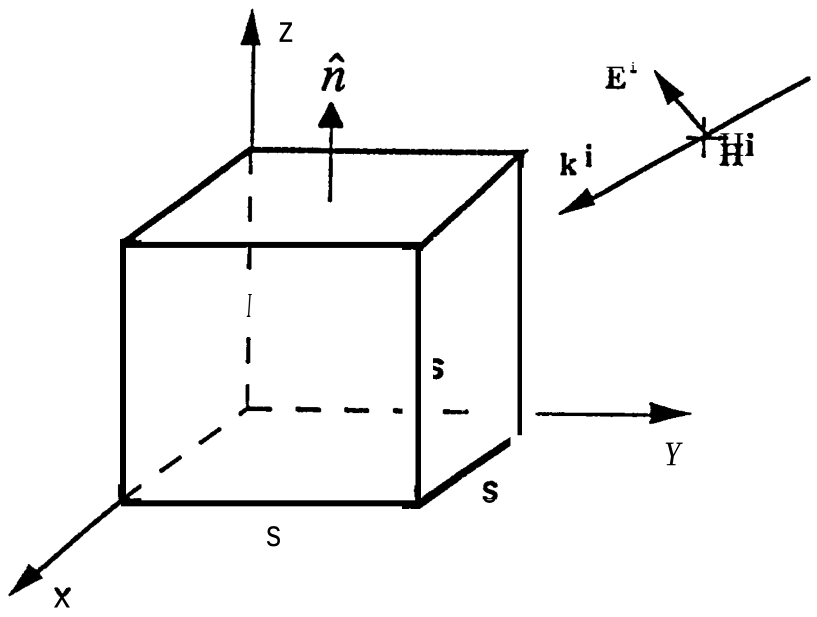

Figure 1. Geometry of cube scatterer of sides. Surface of scatterer is denoted S.

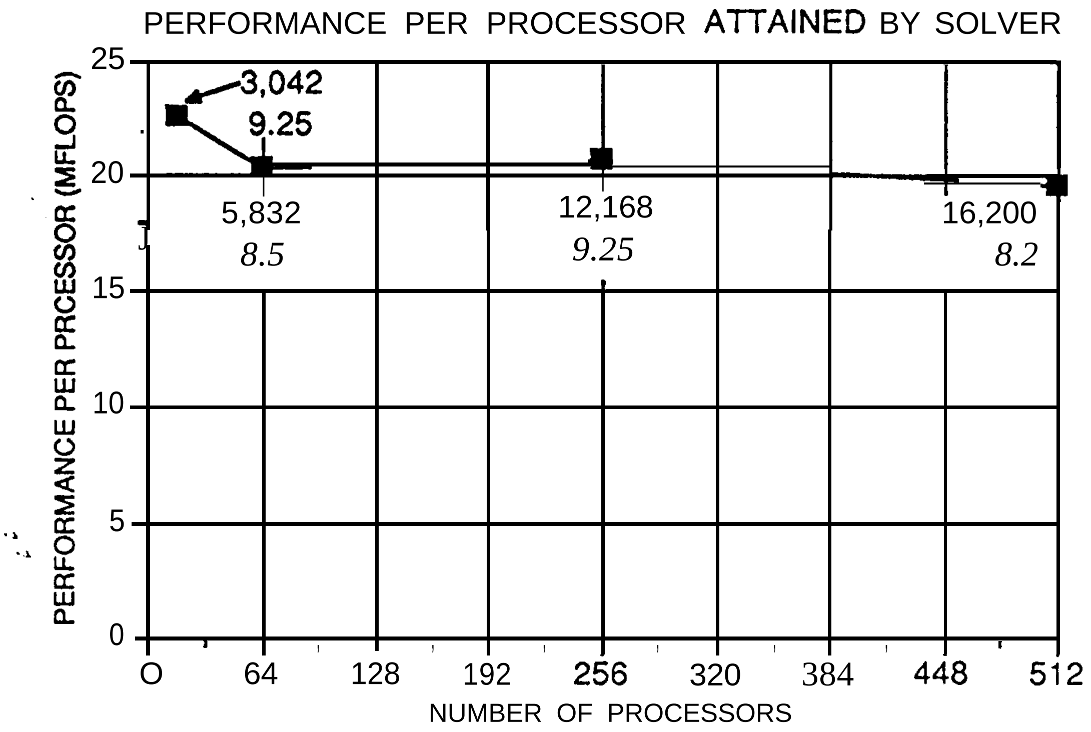

Figure 2. Performance per processor as a function of the number of processors. At each data point, the matrix size is given, as well as the number of million bytes used on each processor to store the matrix.

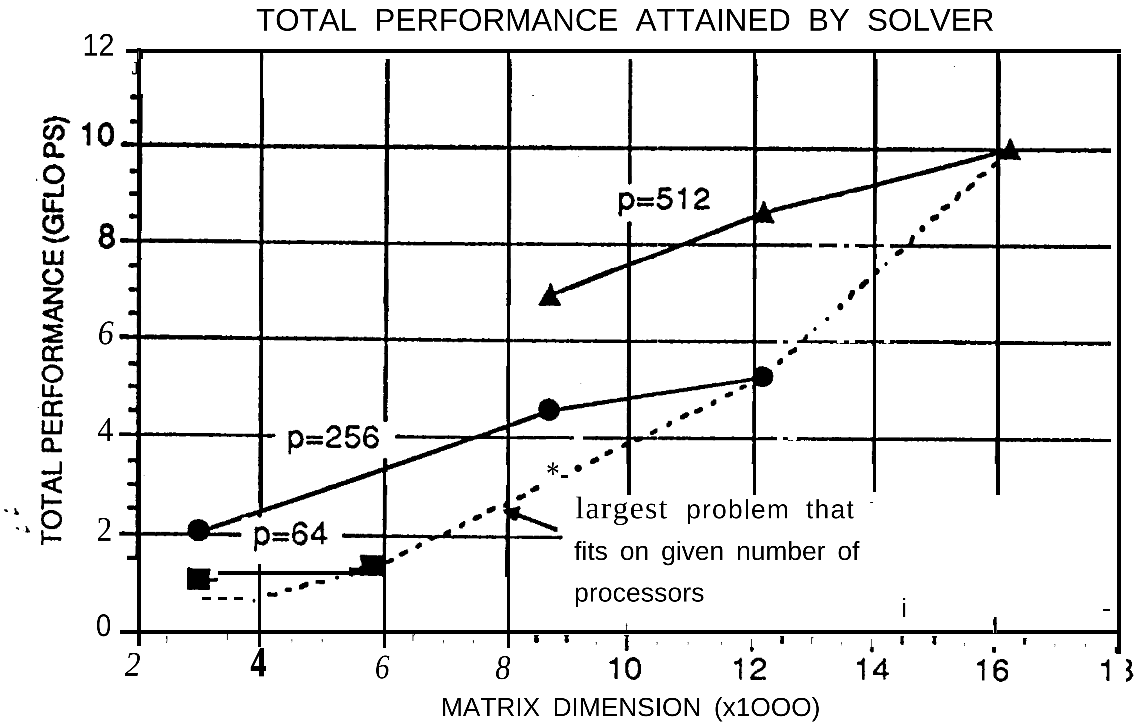

Figure 3. Total performance of solver on different size machines (p is the number of processors in use). Dotted line is performance for scaled problem, i.e., largest size problem that fits on given machine size.

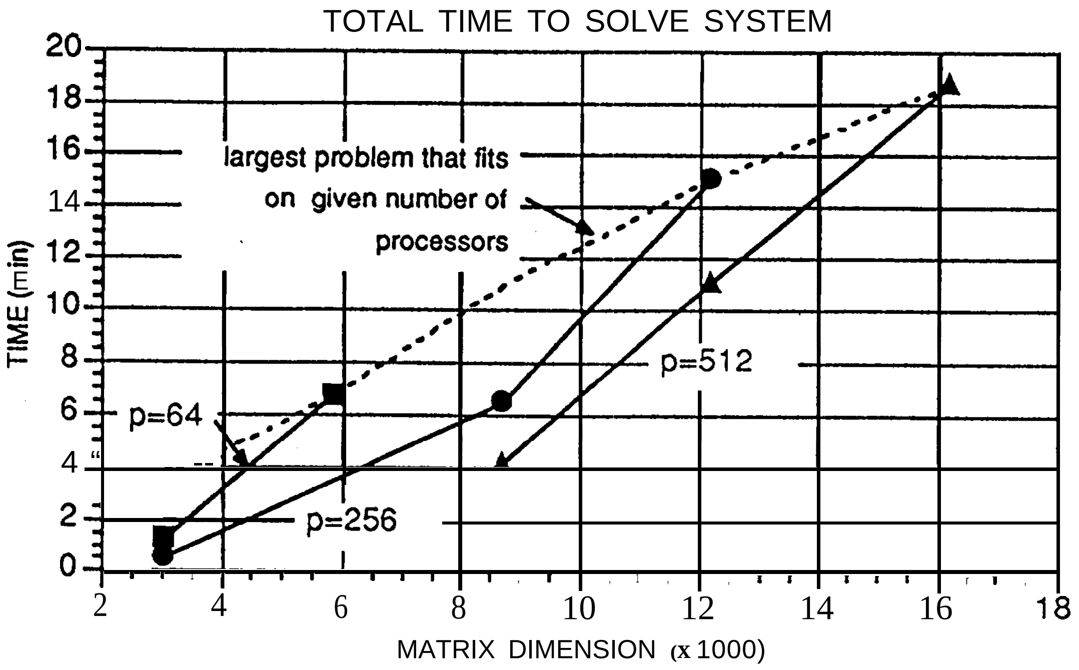

Figure 4. Total time to solve system on different sized machines (p is the number of processors in use). Dotted line is time for scaled problem, i.e., largest size problem that fits on given machine size.

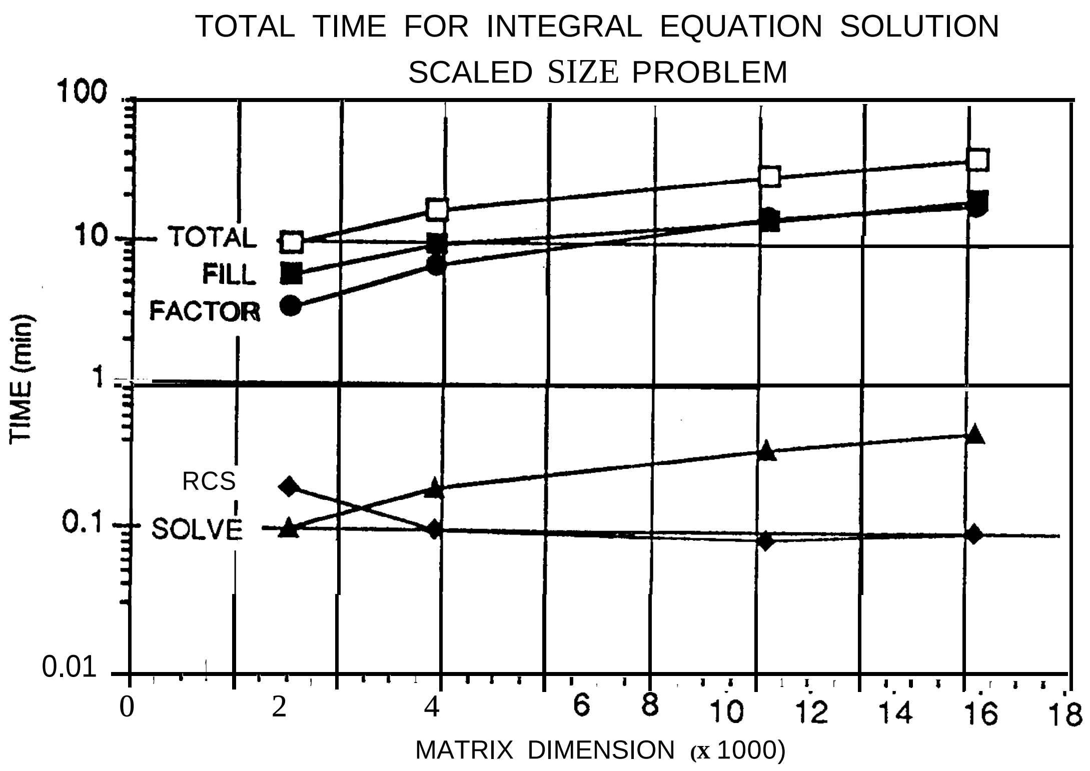

Figure 5. Total time for integral equation solution for scaled problem, i.e., largest size problem that fits on given machine size.

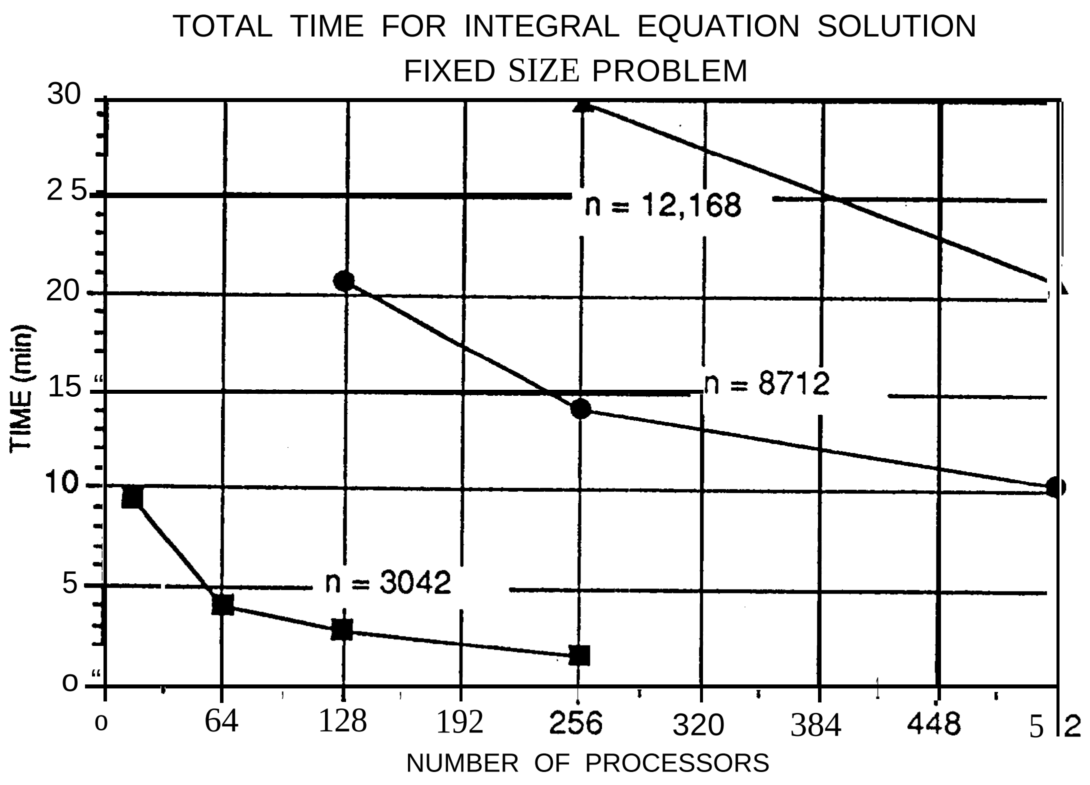

Figure 6. Total time for integral equation solution for fixed size problem.

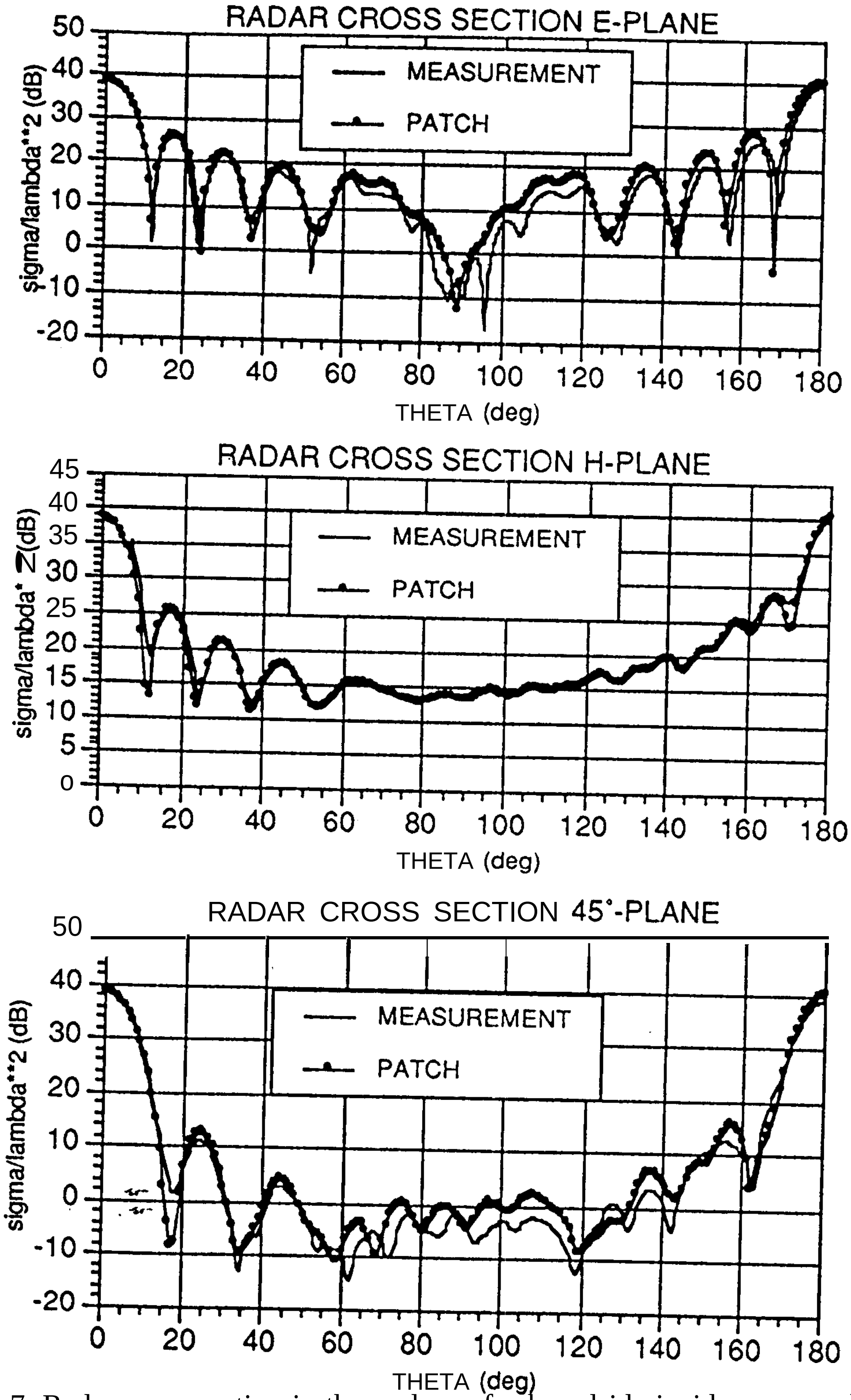

Figure 7. Radar cross section in three planes for broadside incidence on cube of side 5 wavelengths. 16,200 unknowns were used to model the induced current on this object.